Spreadsheet Table Transformations from Examples

William R. Harris *
Dept. of Computer Sciences
University of Wisconsin, Madison
Madison, WI, USA
wrharris@cs.wisc.edu

Sumit Gulwani
Microsoft Research
Redmond, WA, USA
sumitg@microsoft.com

Abstract

Every day, millions of computer end-users need to perform tasks over large, tabular data, yet lack the programming knowledge to do such tasks automatically. In this work, we present an automatic technique that takes from a user an example of how the user needs to transform a table of data, and provides to the user a program that implements the transformation described by the example. In particular, we present a language of programs TableProg that can describe transformations that real users require. We then present an algorithm ProgFromEx that takes an example input and output table, and infers a program in TableProg that implements the transformation described by the example. When the program is applied to the example input, it reproduces the example output. When the program is applied to another, potentially larger, table with a "similar" layout as the example input table, then the program produces a corresponding table with a layout that is similar to the example output table. A user can apply ProgFromEx interactively, providing multiple small examples to obtain a program that implements the transformation that the user desires. Moreover, ProgFromEx can help identify "noisy" examples that contain errors.

To evaluate the practicality of TableProg and ProgFromEx, we implemented ProgFromEx as a module for the Microsoft Excel spreadsheet program. We applied the module to automatically implement over 50 table transformations specified by end-users through examples on online Excel help forums. In seconds, ProgFromEx found programs that satisfied the examples and could be applied to larger input tables. This experience demonstrates that TableProg and ProgFromEx can significantly automate the tasks over tabular data that users need to perform.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; I.2.2 [*Artificial Intelligence*]: Program Synthesis

General Terms Languages, Algorithms, Human Factors

Keywords Program Synthesis, End-user Programming, Programming by Example, Spreadsheet Programming, Table Manipulation, User Intent

* This work was performed during an internship at Microsoft Research, Redmond.

PLDI'11, June 4–8, 2011, San Jose, California, USA.

1. Introduction

More and more end users now apply computers to perform complex and critical tasks on digital data. While users often clearly understand the task that they need a computer to perform, the tools that users may apply to communicate their task to a computer remain limited. At one extreme, graphical user interfaces (GUI's) are highly accessible. However, GUI's typically do not allow users to customize or personalize their program such that the user can fully automate their task. Furthermore, as programs with GUI's support more and more features, users often struggle to discover the features. On the other extreme, general programming languages serve as a fully expressive medium for communicating a task to a computer. However, even the most accessible scripting language requires an amount of time and energy that a typical user is not prepared, and should not be expected, to invest.

Unlike traditional programming, the techniques of *end-user programming* by demonstration or by example [3, 16] allow users to describe a computation via partial execution traces or final outputs resulting from a small set of inputs, without requiring the user to describe how to perform the computation in general. A tool based on such techniques then produces for the user a general program that, given the example inputs, produces the corresponding outputs. Moreover, the program is defined for many more inputs than the examples, so users can apply the program to other inputs to automatically obtain the outputs that they require. Programming by example is an attractive approach, as users often naturally use examples to express the tasks that they need to perform. On help forums that we studied, novice users express tasks through examples so frequently that it is practically a forum convention.

Previous work in programming by example has focused primarily on inferring programs that transform strings of text [9, 14]. However, techniques based on this work cannot yet be used directly by real end users, as in practice, users need to transform rich, semi-structured data. In particular, millions of users every day need to transform tabular data stored in spreadsheets using office programs such as Microsoft Excel [5] or OpenOffice [18]. Often, such transformations do not change the textual content of any cell, or only change the content in a simple way. However, the transformation rearranges the layout of the table, i.e. the manner in which cells are spatially arranged or grouped. Existing techniques for programming by example cannot be applied to infer programs that implement such transformations.

Inferring transformations over spreadsheet tables presents unique challenges compared to the problem of inferring views over relational tables [4]. In particular, a spreadsheet table is in general only a two-dimensional array of cells, where each cell contains a string. Unlike relational tables, rows of a spreadsheet tables are ordered, different rows may contain data with different semantic relationships, and the spreadsheet table may not provide names of column fields. Spreadsheet tables also often have special layout or format-

Example input table:

	Qual 1	Qual 2	Qual 3
Andrew	01.02.2003	27.06.2008	06.04.2007
Ben	31.08.2001		05.07.2004
Carl		18.04.2003	09.12.2009

Example output table:

Andrew	Qual 1	01.02.2003
Andrew	Qual 2	27.06.2008
Andrew	Qual 3	06.04.2007
Ben	Qual 1	31.08.2001
Ben	Qual 3	05.07.2004
Carl	Qual 2	18.04.2003
Carl	Qual 3	09.12.2009

Figure 1. Example input and output tables from an online Excel help forum thread "Using a macro to extract and rearrange data."

ting attributes, such as sub-headers, footers, filler cells (blank cells or cells with some special characters to aid visual readability of table content) [1]. In this paper, we only consider the problem of inferring transformations over spreadsheet tables. All references to "tables" refer to spreadsheet tables, unless otherwise noted.

Example 1. *The tables in Fig. 1 are an example of a common table transformation that a user would like to perform automatically. The tables are from a real Excel help forum thread.* [1] *The thread was started by a novice user, who needed to transform a large table in a given layout into a table in a different layout. To express their transformation, the user provided a small, representative input table, along with the output table that should result from applying the transformation to the input table. Both tables are in Fig. 1. The example input table contains a set of dates on which tests where given, where each date is in a row corresponding to the name of the test taker, and in a column corresponding to the name of the test. For every date, the user needs to produce a row in the output table containing the name of the test taker, the name of the test, and the date on which the test was taken. If a date cell in the input table is empty, then no corresponding row should be produced in the output table. The rows should be produced in the order that the dates are ordered in the input table by row-major order.*

In the help thread, the novice user described their required transformation in a few paragraphs of English, but the novice also included the example input and output tables in Fig. 1 (the output table has been simplified slightly for illustrative purposes; the full implementation of our technique can infer a program for the original example). In about half an hour, an expert user responded with a link to an Excel macro that the expert suspected would be useful for the novice. Twenty minutes after the expert responded, the novice confirmed that the macro implemented the transformation that he or she required.

In this paper, we describe a method for inferring table transformations from examples by applying a general research methodology for designing systems that support programming by example [8]. The steps of this general methodology are:

1. Identify a domain of data on which a large class of users struggle to perform repetitive operations that they can clearly describe with examples.
2. Design a programming language that describes a large proportion of operations that users need to perform on the data domain in practice.
3. Design an algorithm that efficiently infers programs in the language from example inputs and outputs.

In previous work, we applied the above methodology to infer programs that transform strings of text [9]. In this work, we apply the methodology to infer programs that transform tables.

In this paper, we first present a language of programs that implement transformations over tables. While the language cannot describe all transformations over tables, we designed it by studying the transformations that users require in practice, and defining a language that can describe combinations of the most common transformations.

We then present an algorithm that takes an example input and output table and automatically infers in seconds a program in the language that implements a transformation that satisfies the example. If the program is applied to the example input, then the program produces the example output, and if the program is applied to another table with a layout similar to that of the example input, then the program produces a corresponding table with a layout similar to the example output table. End users can apply our language and algorithm to automatically obtain programs that transform multiple, huge tables. To do so, they construct small, representative input and output example tables, and give the example tables to our inference algorithm. The inference algorithm then infers a program which the user can apply to their large input tables. The user in Ex. 1 can apply our algorithm to the example tables in Fig. 1 and in seconds obtain a program that implements their desired table transformation.

The algorithm is highly scalable because it divides the problem of inferring a program that transforms an entire table into subproblems of inferring programs that transform subtables of the original table. It then infers simple programs that transform the subtables, and efficiently combines the simple programs to construct a program that transforms the original table.

On some examples, our algorithm may infer a program that satisfies the input and output pair given by the user, but does not implement the general transformation that the user requires. In this case, the user can refine the inferred program by providing a larger, more descriptive input-output example that demonstrates the behavior on which the original program behaves incorrectly, or by providing multiple input and output examples that together describe the required behavior.

In this paper, we make the following contributions:

1. We present a language of programs, TableProg, that can express a rich set of practical transformations over tabular data. TableProg is designed to express table transformations required by real users, but is conceptually simple and is described by a small semantics.
2. We present a novel algorithm, ProgFromEx, for inferring TableProg programs from example input and output tables. We show the correctness of ProgFromEx, and analyze its complexity.
3. We report our experience using ProgFromEx. We implemented ProgFromEx as a plug-in module for the Microsoft Excel spreadsheet program, and applied the module to automatically infer TableProg programs that implement over 50 transformations specified by examples in online Excel help forums.

The rest of this paper is organized as follows. In §2, we use the example in Fig. 1 to illustrate the challenges of inferring table transformations, the structure of programs in TableProg, and how ProgFromEx infers a program that satisfies the example. In §3, we present TableProg in detail, and in §4, we present ProgFromEx in detail. In §5, we report our experience applying ProgFromEx

[1] http://www.excelforum.com/excel-programming/698490-using-a-macro-to-extract-and-rearrange-data.html

to infer table programs from real-world examples. In §6 we discuss related work, and in §7, we conclude.

2. Overview

Users often need to transform the layout of tables in non-trivial ways. In this section, we use the running example in Fig. 1 to observe in more detail the challenges in inferring a transformation from an example input and output. From these observations, we motivate the design of our language TableProg of table programs and our algorithm for inferring TableProg programs.

2.1 An Example Table Program

Consider a program that, given the example input table from Fig. 1, produces the example output table (i.e. a program that *satisfies* the examples). One insight into a possible structure of such a program is guided by the following property of example tables, which we have observed to hold over almost all example tables given by real users.

Remark 1. Often, a subset of the cells in an output table (i.e. a *substructure* of the output table) can be produced by treating the cells in the input table as a sequence ordered by row-major order, selecting some cells in the sequence, and spatially rearranging the selected cells while preserving their row-major order.

Guided by Remark 1, we initially suppose that a table program makes a series of passes over the input table in row-major order. In each pass, the program selects a subset of cells in the input table, and produces cells in the output table that hold the same text as the selected cells, though they may be produced in a different spatial arrangement. We call each pass a *filter program*, and say that a filter program reads the cells of the input table in row-major order, and checks if each cell satisfies a *mapping condition*. If an input cell satisfies the mapping condition, then the filter program produces a new cell in the output table that contains the same text as the input. The new cell is produced at the bottom of a column determined by an *output sequencer*.

Example 2. *Remark 1 can be applied to the tables in Fig. 1 to derive a filter program* F *that produces column 3 (we adopt the convention of most spreadsheet programs and describe tables using 1-based indexing). To produce column 3, a filter program passes over the cells in the input table in row-major order, and checks each cell against the mapping condition that the cell is not in row 1, not in column 1, and is non-empty. If the input cell satisfies the mapping condition, then the filter program's output sequencer determines that filter program should add the new cell to column 3.*

Ex. 2 illustrates that filter programs are often powerful enough to produce substructures of an output table. However, Fig. 1 demonstrates that filter programs on their own are often insufficient or impractical for producing an entire output table. First, to produce column 1 of the output table, a filter program would need to perform complex reasoning. When the filter program would check the input cell containing "Andrew," it would add cells holding this text to column 1 of the output three times in sequence. However, when the filter program would check the cells with texts "Ben" and "Carl", it would only add cells to column 1 of the output two times each. To produce column 2 of the output table, a filter program would need to apply different complex reasoning. Such a program would check the input cells containing the texts "Qual k," and would produce column 2 of the output table by interleaving multiple cells with these texts. Intuitively, it seems difficult to derive a language of filter programs that can perform such reasoning in a single pass over the input table. To derive programs that can produce such substructures of the output, we apply a new observation.

Remark 2. Two cells in an output table that are in the same row or column often hold the same text as two cells in the input table that are in the same row or the same column.

Example 3. *For Fig. 1, let cell c be any cell in column 1 of the output table, and let cell d be in column 3 of the same row as c. Then c and d hold the same text as cells in columns 1 and 3 of some row of the input table.*

To apply Remark 2, we first enrich our notion of a filter program. Above, we described a filter program as taking a table as input and producing a substructure of a table as output. We now describe a filter program as taking a table as input, and computing a *map* from coordinates of cells in the input table to the coordinates of cells that will be in the output table. In general, a map may be an arbitrary binary relation pairing input and output coordinates; it need not be a function. A table program applies a coordinate map to an input table to produce a substructure of the output table; for every entry (c, d) in the map, the table program produces a new cell in the output table at coordinate d, and fills the cell with the text in the input table at c. If a coordinate map maps every cell that will be in a column k of an output table, then we say that the map *maps to column* k.

Example 4. *In Ex. 2, we described a filter program* F *as producing column 3 of the output table. If we redefine* F *as computing a map m_{F} from coordinates of cells in the input table to coordinates of cells that will be in column 3 of the output table, then m_{F} for the example tables in Fig. 1 is the following set of pairs of input and output coordinates:*

$$m_{\mathsf{F}} = \left\{ \begin{array}{l} ((2,2),(1,3)), ((2,3),(2,3)), ((2,4),(3,3)), \\ ((3,2),(4,3)), ((3,4),(5,3)), \\ ((4,3),(6,3)), ((4,4),(7,3)) \end{array} \right\}$$

By defining filter programs to compute maps between coordinates, we can build *associative programs* from filter programs. Like a filter program, an associative program computes a map from coordinates of cells in the input table to coordinates of cells to be in the output table. However, to do so, an associative program uses a filter program F to compute an initial map between coordinates, and then alters each pair of coordinates in the map to produce its own map. The associative program alters the map computed by F by applying a *relative function* R_1 to each input coordinate to obtain a new input coordinates, and applying another relative function R_2 to each output coordinate to obtain new output coordinates. We denote such an associative program as $(\mathsf{F}, \mathsf{R}_1, \mathsf{R}_2)$.

Example 5. *For Fig. 1, an associative program* A_1 *maps to column 1 of the output table.* A_1 *first uses* F *to compute the map m_{F} from Ex. 4.* A_1 *alters the input coordinates of m_{F} by applying to each input coordinate a relative function* $\texttt{RELCOL}_1$ *that computes the coordinate in the same row and in column 1.* A_1 *alters the output coordinates of m_{F} by applying* $\texttt{RELCOL}_1$ *to each output coordinate. The resulting map m_{A_1} maps coordinates in column 1 of the input table to coordinates of cells to be produced in column 1 of the output table:*

$$m_{\mathsf{A}_1} = \left\{ \begin{array}{l} ((2,1),(1,1)), ((2,1),(2,1)), ((2,1),(3,1)), \\ ((3,1),(4,1)), ((3,1),(5,1)), \\ ((4,1),(6,1)), ((4,1),(7,1)) \end{array} \right\}$$

A second associative program A_2 *maps to column 2 of the output table. Like* A_1*,* A_2 *first uses* F *to compute m_{F}.* A_2 *alters each input coordinate in m_{F} by applying a relative function* $\texttt{RELROW}_1$ *that computes the coordinate in the same column but in row 1.* A_2 *alters each output coordinate in m_{F} by applying a relative function* $\texttt{RELCOL}_2$*, where* $\texttt{RELCOL}_2$ *is defined analogously to* $\texttt{RELCOL}_1$*. The resulting map m_{A_2} maps coordinates in row 1 of the input to*

coordinates of cells to be produced in column 2 of the output:

$$m_{\mathsf{A}_2} = \left\{ \begin{array}{l} ((1,2),(1,2)),((1,3),(2,2)),((1,4),(3,2)), \\ ((1,2),(4,2)),((1,4),(5,2)), \\ ((1,3),(6,2)),((1,4),(7,2)) \end{array} \right\}$$

An associative program may use a map computed by a filter program, but may in general also use a map computed by another associative program. This is because an associative program only uses a filter program to compute a map over table coordinates, but associative programs themselves compute maps over table coordinates. We thus refer to both filter programs and associative programs as *component programs*, and say that in general, an associative program can be built from a component program.

A table program P built from filter and associative programs can take the example input table from Fig. 1 and produce the example output table. P computes the coordinate map of the filter program F described in Ex. 4, the map of the associative program A_1, and the map of A_2 described in Ex. 5, and then applies the coordinate maps to produce a table. When P is applied to the example input from Fig. 1, it produces the example output. P is a program in the language TableProg, which is presented formally in §3.

2.2 Inferring the Example Table Program

The table program P described in §2.1 satisfies the example tables from Fig. 1. The algorithm ProgFromEx, when given the example tables from Fig. 1, automatically infers P. Like P, each table program is a set of two different types of component programs: filter programs and associative programs. Thus ProgFromEx infers a table program in two steps, building component programs of a particular type in each step. In step 1), ProgFromEx builds a set of filter programs. In step 2), ProgFromEx iteratively builds associative programs from component programs that it has already built until it finds a set of component programs that map to all cells in the example output table.

In step 1), ProgFromEx infers a set of filter programs from a fixed set of *candidate-map rules*. Each candidate-map rule, given example input and output tables, produces a set of *consistent* maps between coordinates in the example tables. A map is consistent for an example input and output if, when applied to the example input, the map produces a substructure of the example output. Unlike a filter program, candidate-map rules cannot compute a map from only an input table. They are applied to an example input and output to suggest a map that a filter program may compute when the program is applied to the example input. Thus we call such a map a *candidate map*. From a candidate map, ProgFromEx infers a general filter program that computes the candidate map when applied to the example input, but also computes analogous maps when applied to other input tables.

Example 6. ProgFromEx *uses a candidate-map rule to suggest the coordinate map m_{F} from Ex. 4.* ProgFromEx *is given candidate-map rules that specify: "map a coordinate in the input table to a coordinate in column k of the output table if and only if the values at those coordinates are equal," with one such rule for each column k in the example output table. The rule for $k = 3$ generates a candidate map from each coordinate holding a date in the input table to the coordinate in the output table holding the same date. This map is m_{F}.*

Given a candidate map, ProgFromEx attempts to infer a filter program that computes the map when the program is applied to the example input. To infer a filter program, ProgFromEx must first infer the filter program's mapping condition. For a filter program to compute the candidate map, the program's mapping condition must be satisfied by every input cell mapped by the candidate map, and must not satisfied by any input cell that is not mapped by the candidate map. ProgFromEx infers the mapping condition as a conjunction over a fixed set of atomic predicates and their negations using a greedy algorithm, discussed in §4.1. Each atomic predicate describes some feature of a cell.

Example 7. ProgFromEx *infers a filter program to implement the candidate map m_{F} suggested by a candidate-map rule in Ex. 6. To infer a filter program,* ProgFromEx *first infers the filter program's mapping condition. The mapping condition must be satisfied by all of the cells in the example input that are mapped by m_{F}, and must not be satisfied by any of the cells in the example input that are not mapped by m_{F}. Suppose that* ProgFromEx *is given a predicate that decides if a cell is in row 1, a predicate that decides if a cell is empty, and for each column k in the example input table, a predicate that decides if a cell is in column k. Given these predicates,* ProgFromEx *infers the conjunctive mapping condition stated informally in Ex. 2.*

To infer a filter program that computes a candidate map, ProgFromEx must also infer an output sequencer for the filter program. To infer an output sequencer, ProgFromEx orders the output coordinates in the candidate map by the order in which the filter program must map them; i.e., ProgFromEx orders the output coordinates by the row-major ordering of the input cells that map to them. ProgFromEx then checks if the ordered sequence of output coordinates matches the output coordinates described by some output sequencer in a fixed set of sequencers. If so, then ProgFromEx builds a filter program by pairing the matching sequencer with the corresponding mapping condition.

Example 8. *For* ProgFromEx *to infer a filter program that computes the candidate map m_{F} from Ex. 6, it must infer an output sequencer that describes the output coordinates mapped to by m_{F}. To infer a sequencer,* ProgFromEx *orders the output coordinates mapped to by m_{F} by the row-major order of the input cells that map to them under m_{F}. The output coordinates ordered in this way form column 3 of the example output table.* ProgFromEx *thus builds a filter program that computes m_{F} from a sequencer that maps to coordinates in column 3, as opposed to a sequencer that maps to coordinates in column 1 or 2.*

In step 1), ProgFromEx generates a set of candidate maps, and infers filter programs that implement the candidate maps; the inferred filter programs serve as an initial set of component programs for step 2) In step 2), ProgFromEx iteratively builds associative programs from the component programs that it has already built. As described in §2.1, an associative program is built from a component program and two relative functions. ProgFromEx thus builds associative programs by combining component programs that it has already built with relative functions drawn from a fixed set. If the resulting associative program computes a map that is consistent for the example tables, then ProgFromEx retains the associative program as a component program from which to build more associative programs.

Example 9. ProgFromEx *builds the associative programs A_1 and A_2 described in Ex. 5 from the filter program F described in Ex. 2.* ProgFromEx *finds A_1 by combining filter program F and relative functions $\texttt{RELCOL}_1$ to build the associative program $\mathsf{A}_1 = (\mathsf{F}, \texttt{RELCOL}_1, \texttt{RELCOL}_1)$. A_1 is consistent with the example input and output table, so* ProgFromEx *retains it as a component program. Similarly,* ProgFromEx *finds A_2 from Ex. 5 by combining F and the relative functions $\texttt{RELROW}_1$ and $\texttt{RELCOL}_1$ to build the associative program $\mathsf{A}_2 = (\mathsf{F}, \texttt{RELROW}_1, \texttt{RELCOL}_1)$. A_2 is also consistent with the example tables, so* ProgFromEx *retains it as a component program.*

ProgFromEx *may also build other associative programs, such as $(\mathsf{F}, \texttt{RELCOL}_1, \texttt{RELCOL}_2)$ and $(\mathsf{F}, \texttt{RELROW}_1, \texttt{RELCOL}_1)$. However,*

$$
\begin{aligned}
\text{TableProg} &:= \texttt{TABPROG}(\text{CompProg}_1, \ldots, \text{CompProg}_n) \\
\text{CompProg} &:= \text{FilterProg} \mid \text{AssocProg} \\
\text{FilterProg} &:= \texttt{FILTER}(\text{MapCond}, \texttt{SEQ}_{i,j,k}) \\
\text{MapCond} &:= \texttt{AND}(\text{MapPred}_1, \text{MapPred}_2, \ldots, \text{MapPred}_n) \\
\text{MapPred} &:= \texttt{ROWEQ}(\texttt{TERM}_1, \texttt{TERM}_2) \mid \texttt{COLEQ}(\texttt{TERM}_1, \texttt{TERM}_2) \\
&\quad \mid \texttt{DATAEQ}(\texttt{TERM}_1, \texttt{TERM}_2) \mid \texttt{NOT}(\text{MapPred}) \\
\text{AssocProg} &:= \texttt{ASSOC}(\text{CompProg}, \text{RelFunc}_1, \text{RelFunc}_2) \\
\text{RelFunc} &:= \texttt{RELCOL}_i \mid \texttt{RELROW}_i
\end{aligned}
$$

Figure 2. The syntax of TableProg.

ProgFromEx *determines that these associative programs are not consistent with the example input and output, and thus does not retain them as component programs.*

In steps 1) and 2) described above, ProgFromEx infers component programs that map from coordinates of an example input to coordinates of an example output. ProgFromEx must determine when it has inferred enough component programs to produce a table program that satisfies the example. A table program is a set of component programs that map to coordinates of cells to be produced in the output table. A set of component programs forms a table program that satisfies an example if every component program in the set is consistent with the example, and for every coordinate in the example output table, some component program in the set maps to the coordinate.

Example 10. ProgFromEx *finds a set of component programs that map to all coordinates in the output table after executing step 1) and one iteration of step 2). In step 1),* ProgFromEx *finds the filter program* F *described in Ex. 2, which maps to column 3 of the example output table. In one iteration of step 2),* ProgFromEx *finds the associative program* $\mathsf{A}_1 = (\mathsf{F}, \texttt{RELCOL}_1, \texttt{RELCOL}_1)$*, which maps to column 1 of the example output, and the associative program* $\mathsf{A}_2 = (\mathsf{F}, \texttt{RELROW}_1, \texttt{RELCOL}_2)$*, which maps to column 2 of the example output.* ProgFromEx *combines* F*,* A_1*, and* A_2 *to build a table program that maps to all cells of the example output, and thus satisfies the example.*

3. A Language of Table Programs

We now present a language of table programs, TableProg, that can express table transformations required by real users. We first present the syntax of TableProg, and then define the semantics of a program in TableProg as a function from an input table to an output table.

3.1 Syntax of Table Programs

By the formal syntax for TableProg given in Fig. 2, a table program (TableProg) is a set of component programs (CompProg). A component program is either a filter program (FilterProg) or an associative program (AssocProg). A filter program makes a single pass over an input table. During the pass, the filter selects certain cells from the input table and maps them to a substructure of the output table. This is reflected in the syntax of a FilterProg as follows. A FilterProg consists of a mapping condition over states of a filter program (MapCond) and an output coordinate sequencer ($\texttt{SEQ}_{i,j,k}$). The MapCond selects which input coordinates are mapped to the output table, and the $\texttt{SEQ}_{i,j,k}$ for natural numbers i, j, and k defines the output coordinate to which the selected input cell maps. A MapCond is a conjunction of cell predicates (MapPred). Each MapPred is an equality (or disequality) predicate over cell terms. Specifically, a MapPred is an equality predicate either over the row, column, or data in cell TERM's. A cell TERM is either a variable bound to a particular cell (such as the input cell being checked by the filter program) or a constant cell value.

$$
\begin{aligned}
[\![\texttt{TABPROG}(\{\mathsf{C}_i\})]\!] &= \lambda T_I.\, \left\{ \begin{array}{l} (c_2, d) \mid (c_1, d) \in T_I, \\ \qquad (c_1, c_2) \in \bigcup_i \{[\![\mathsf{C}_i]\!](T_I)\} \end{array} \right\} \\
[\![\texttt{FILTER}(\mathsf{G}, \mathsf{S})]\!] &= \lambda T_I.\, \text{FilterIter}_{\mathsf{G},\mathsf{S}}(\text{InitState}) \\
[\![\texttt{AND}(\{\mathsf{L}_i\})]\!] &= \lambda \sigma.\, \bigwedge_{i=1}^{n} [\![\mathsf{P}_i]\!](\sigma) \\
[\![\texttt{ROWEQ}(\mathsf{T}_1, \mathsf{T}_2)]\!] &= \lambda \sigma.\, \left(\begin{pmatrix} \lambda((r_1, c_2), d_1), ((r_2, c_2), d_2). \\ r_1 = r_2 \end{pmatrix} \begin{pmatrix} \\ (\sigma(\mathsf{T}_1), \sigma(\mathsf{T}_2)) \end{pmatrix} \right) \\
[\![\texttt{SEQ}_{i,j,k}]\!] &= \lambda(\mathrm{r}, \mathrm{c}).\, \left(\begin{array}{l} \text{if r} < i \text{ then } (i, j) \\ \text{else if c} < j \text{ then } (r, c+1) \\ \text{else } (r+1, j) \end{array} \right) \\
[\![\texttt{ASSOC}(\mathsf{C}, \mathsf{R}_1, \mathsf{R}_2)]\!] &= \lambda T_I.\, \left\{ \begin{array}{l} ([\![\mathsf{R}_1]\!](r_1, c_1), [\![\mathsf{R}_2]\!](r_2, c_2)) \mid \\ ((r_1, c_1), (r_2, c_2)) \in [\![\mathsf{C}]\!](T_I) \end{array} \right\} \\
[\![\texttt{RELCOL}_i]\!] &= \lambda(\mathrm{r}, \mathrm{c}).(\mathrm{r}, i) \\
[\![\texttt{RELROW}_i]\!] &= \lambda(\mathrm{r}, \mathrm{c}).(i, \mathrm{c}) \\
\text{FilterIter}_{\mathsf{G},\mathsf{S}}(\sigma) &= \begin{pmatrix} \text{if } [\![\mathsf{G}]\!](\sigma) \\ \text{then } \{(\sigma(\text{CurIn}), \sigma(\text{CurOut}))\} \text{ else } \emptyset \end{pmatrix} \\
&\quad \cup \begin{pmatrix} \text{if IsLastCell}(\sigma(\text{CurIn})) \text{ then } \emptyset \\ \text{else FilterIter}_{\mathsf{G},\mathsf{S}}(\text{IterUpdate}_{\mathsf{S}}(\sigma)) \end{pmatrix} \\
\text{IterUpdate}_{\mathsf{S}}(\sigma) &= \sigma \begin{bmatrix} \text{CurIn} \leftarrow \text{NextInCoord}(\sigma), \\ \text{CurOut} \leftarrow [\![\mathsf{S}]\!](\sigma) \end{bmatrix}
\end{aligned}
$$

Figure 3. The semantics of TableProg.

An associative program AssocProg is built from a component program CompProg and two relative functions RelFunc_1, RelFunc_2. A relative function can be $\texttt{RELCOL}_i$ or $\texttt{RELROW}_i$, where i is a fixed natural number.

3.2 Semantics of Table Programs

We now present the semantics of TableProg. The semantics of TableProg is defined formally in Fig. 3 by the semantic function $[\![\cdot]\!]$ that interprets syntactic forms of TableProg as semantic values. The domain of semantic values is defined as follows. Let a *coordinate* (r, c) with $r, c \in \mathbb{N}$ be an ordered pair built from a row and column number, and let a *cell* $((r, c), d)$ be an ordered pair built from a coordinate (r, c) and data string d. A table T is a set of cells. A table program $\mathsf{P} = \texttt{TABPROG}(\{\mathsf{C}_i\}_i)$ is a function from a table to a table. Each component program C_i is interpreted as a partial map from coordinates of cells in the input table to coordinates of cells that will be produced in the output table. For each cell $((r, c), d)$ in the input table with (r, c) a coordinate in the domain of some map $[\![\mathsf{C}_i]\!]$, P produces a cell $([\![\mathsf{C}_i]\!](r, c), d)$ in the output table.

Every component program is either a filter program or an associative program. A filter program $\texttt{FILTER}(\mathsf{G}, \mathsf{S})$ maps coordinates of cells in the input table to output coordinates by checking each cell in the input table in a fixed order, such as row-major order. In Fig. 3, this order is defined by a constant InitState that defines the coordinate of the first input cell (e.g. $(0, 0)$ in row-major order), a predicate IsLastCell that decides if a coordinate is the last in the order, and a function NextInCoord from input coordinates to input coordinates that takes an input coordinate and computes the next coordinate in the order. As the filter program checks input cells, it maintains a state σ, which distinguishes certain key cells,

such as the current input cell (CurIn) and the current output cell (CurOut), by binding the cells to corresponding variables. When the filter program checks each cell of the input table, it updates σ so that the variable CurIn points to the cell to be checked. The filter program then checks if σ satisfies the filter's mapping condition, $\mathsf{G} = \texttt{AND}(\{\mathsf{L}_i\}_i)$. A state σ satisfies G if and only if σ satisfies every literal L_i. The semantics of each literal is standard; Fig. 3 gives the semantics of the predicate $\texttt{ROWEQ}(\texttt{TERM}_1, \texttt{TERM}_2)$ as an example. Whether or not σ satisfies a predicate is decided by the values in σ of cell terms, such as the variables CurIn and CurOut. If σ satisfies the mapping condition G, then the filter program maps the current input coordinate, which is bound to CurIn, to the current output coordinate, which is bound to CurOut.

If the filter program maps the current input coordinate, it updates the coordinate of the current output cell according to the filter program's output sequencer S. Whenever an output sequencer $\mathsf{S} = \texttt{SEQ}_{i,j,k}$ is applied, it updates the current output coordinate to be the next coordinate in the output table by row major order that is at or below row i and between columns j and k. Because j and k are fixed, such a sequencer can be applied by a filter program to produce columns with an unbounded number of rows, but it cannot be applied to produce an unbounded number of columns. In this paper, we assume that an example output table has the same, fixed number of columns as all tables that the user expects the table program to produce. However, we can extend TableProg to infer programs that produce an unbounded number of columns by extending the set of output sequencers.

Like a filter program, an associative program $\mathsf{A} = \texttt{ASSOC}(\mathsf{C}, \mathsf{R}_1, \mathsf{R}_2)$ maps coordinates in the input table to coordinates of cells to be produced in the output table. A maps coordinates by first computing the map m_C of its component program C. From m_C, A computes its own map by applying the relative function R_1 to each input coordinate in m_C, and by applying the relative function R_2 to each output coordinate in m_C. A relative function $\texttt{RELCOL}_i$ takes a coordinate and computes the coordinate in the same row, but in column i, where i is a fixed constant. A relative function $\texttt{RELROW}_i$ takes a coordinate and computes the coordinate in the same column, but in row i. In this way, an associative program A computes a coordinate map by altering the coordinate map of a component program C.

Example 11. *The table program* P *from Ex. 10 is represented formally as follows. Let* $\texttt{CONSTCELLCOL}(n)$ *be a cell at column* n*, let* $\texttt{CONSTCELLROW}(n)$ *be a cell at row* n*, and let* $\texttt{CONSTCELLDATA}(d)$ *be a cell with data* d*. Let*

$$\mathsf{G} = \texttt{AND}\begin{pmatrix}\texttt{NOT}(\texttt{ROWEQ}(\texttt{CURCELL}, \texttt{CONSTCELLCOL}(1))),\\ \texttt{NOT}(\texttt{COLEQ}(\texttt{CURCELL}, \texttt{CONSTCELLROW}(1))),\\ \texttt{NOT}(\texttt{DATAEQ}(\texttt{CURCELL}, \texttt{CONSTCELLDATA}(\text{""})))\end{pmatrix}$$

$$\mathsf{F} = \texttt{FILTER}(\mathsf{G}, \texttt{SEQ}_{1,3,3})$$

The table program is then

$$\texttt{TABPROG}\begin{pmatrix}\mathsf{F}, \texttt{ASSOC}(\mathsf{F}, \texttt{RELCOL}_1, \texttt{RELCOL}_1),\\ \texttt{ASSOC}(\mathsf{F}, \texttt{RELROW}_0, \texttt{RELCOL}_2)\end{pmatrix}$$

4. Inferring Table Programs from Examples

We now give an algorithm ProgFromEx that, given example input and output tables, infers a TableProg program that satisfies the examples. We then claim that ProgFromEx is correct, and analyze its performance.

Input: example input table T_I, example output table T_O,
Output: TableProg program InferredProg, where InferredProg$(T_I) = T_O$, or unmapped output table.

```
/* Step 1): collect filter programs.                    */
1  for CandMap ∈ EnumCandMaps(T_I, T_O, CandRules) do
2      MapCond ← CondFromMap(CandMap, StatePreds) ;
3      OutCoordSeq ← SeqFromMap(CandMap, Seqs) ;
4      FilterProgram ← FILTER(MapCond, OutCoordSeq) ;
5      FilterPrograms ← AddDistMap(FilterPrograms,
       FilterProgram) ;
6  end
   /* Step 2): collect associative programs.            */
7  Comps ← ∅ ;
8  Worklist ← FilterPrograms ;
9  while NewComps ≠ ∅ do
10     CompProg ← Choose( Worklist) ;
11     Worklist ← Worklist \{CompProg} ;
12     for Rel_1, Rel_2 ∈ RelFuncs do
13         AssocPrg ← ASSOC(CompProg, Rel_1, Rel_2) ;
14         if IsConsistent(AssocPrg, T_I, T_O) and
           Map(AssocPrg) ∉ Maps( Comps ∪ Worklist) then
15             Worklist ← AddDistMap(Comps, AssocPrg) ;
16         end
17     end
18 end
19 if IsOnto(Maps(Comps)) then
20     return TABPROG(Comps) ;
21 else
22     return UnmappedOutput(Comps) ;
23 end
```

Figure 4. The inference algorithm ProgFromEx.

4.1 An Inference Algorithm for TableProg

ProgFromEx, given example input and output tables, infers a table program in TableProg that satisfies the example. ProgFromEx infers programs "bottom-up," in that it iteratively collects a set of component programs that may be combined to form a table program. If ProgFromEx finds a set of component programs that form a table program that satisfies the example, then ProgFromEx returns the table program. If ProgFromEx cannot find such a table program, then ProgFromEx provides to the user the substructure of the output table to which no component program maps.

ProgFromEx, given in Fig. 4, takes from the user an example input table T_I and example output table T_O. ProgFromEx also is defined over four fixed sets of objects: a set CandRules of candidate-map rules, a set StatePreds of predicates for mapping conditions, a set Sequencers of output sequencers, and a set RelativeFuncs of relative functions. These sets are fixed, perhaps configured by an expert user or administrator.

ProgFromEx finds a table program to satisfy the example in two steps. In step 1), ProgFromEx collects a set of filter programs that map to substructures of T_O (Fig. 4, lines [1]–[6]). To find a set of such filter programs, ProgFromEx applies CollectFilters (line [1]), which first collects a set of *candidate maps* over the example tables by applying the candidate-map rules CandRules.

Definition 1. A *candidate map* is a map from coordinates of cells in T_I to coordinates of cells in T_O. Each candidate map satisfies the following conditions:

1. The candidate map maps input coordinates to output coordinates with equal data (i.e. the candidate map is consistent). For-

mally, if the candidate map contains an entry $((r_1, c_1), (r_2, c_2))$ with $((r_1, c_1), d_1) \in T_I$ and $((r_2, c_2), d_2) \in T_O$, then $d_1 = d_2$.

2. The candidate map maps to coordinates described by an output sequencer. Formally, the candidate map maps to every coordinate in T_O at or below row i between columns j and k of the output table for some i, j, and k.
3. The candidate map preserves row-major order. The sequence of pairs in the candidate map ordered by the row-major ordering of the input coordinates is equal to the sequence of entries ordered by the row-major ordering of the output coordinates.

For each candidate map CandMap generated by CandRules, ProgFromEx attempts to infer a filter program that computes CandMap (lines [2]–[4]). To infer such a filter program, it must infer a mapping condition (line [2]) and an output sequencer (line [3]). To infer a mapping condition, ProgFromEx applies CondFromMap, which computes the states of a hypothetical filter program as it reads, and potentially maps, each cell in the example input table. If in a given state, a filter program reads a cell that is mapped by the candidate map, then let this state be a *read state* of the filter program. For a set of read states RS, CondFromMap constructs the following MapCond, which is the strongest condition that is satisfied by all read states:

$$\texttt{AND}\left(\bigcap_{\sigma \in \mathsf{RS}} \{l \mid p \in \mathsf{StatePreds}, l \in \{p, \texttt{NOT}(p)\}, \sigma(l)\}\right)$$

Where $\sigma(l)$ denotes that the literal l is satisfied in σ. CondFromMap then checks if any non-read state satisfies MapCond. If so, then no conjunction of literals from StatePreds may act as a mapping condition for CandMap. If not, then MapCond acts as a mapping condition for CandMap. For G, ProgFromEx can immediately infer an output coordinate sequencer OutCoordSeq (line [3]) using conditions (2) and (3) from Defn. 1. ProgFromEx then pairs condition MapCond and sequencer OutCoordSeq to build a filter program that computes CandMap (line [4]).

In step 2) (lines [7]–[18]), ProgFromEx uses the filter programs found in step 1) to build associative programs until it can use the set of filter and associative programs to build a table program that satisfies T_I and T_O, or it determines that no such table program exists. ProgFromEx iteratively builds associative programs as follows. Over each iteration of the loop at line [9], ProgFromEx maintains a worklist (Worklist) of component programs that it will use to build more associative programs, and a set of component programs (Comps) from which it has already built associative programs. At the beginning of the first iteration, Worklist is initialized to all of the filter programs found in step 1) (line [8]), and Comps is initialized to be empty (line [7]).

ProgFromEx executes an iteration of step 2) as follows. First, ProgFromEx chooses an element CompProg from its worklist (line [10]). ProgFromEx then builds associative programs from CompProg. An associative program consists of a component program and a pair of relative functions. Thus to build associative programs from a component program CompProg, ProgFromEx enumerates over all pairs of relative functions (line [12]). For relative functions $\mathsf{RelFunc}_1$ and $\mathsf{RelFunc}_2$, ProgFromEx builds the corresponding associative program AssocProg (line [13]). ProgFromEx then decides if AssocProg computes a map that is consistent for T_I and T_O (line [14]). If so, and if the map computed by AssocProg is not computed by any component program in Comps or Worklist, then ProgFromEx adds AssocProg to Worklist (line [15]).

ProgFromEx iteratively builds associative programs until it determines either that it has found a set of component programs that map to all cells in T_O (i.e. a set that *covers* T_O), or determines that it can find no such set of component programs (line [9]). To check if a set of component programs covers T_O, ProgFromEx checks if every coordinate c in T_O is mapped to by some component program in the set. If ProgFromEx finds such a set, then it builds a table program from the set and returns the table program (line [20]). Otherwise, it returns the set of output cells to which no component program maps (line [22]). The user can examine the output cells to understand why ProgFromEx could not infer a program to satisfy the examples, perhaps finding errors or noise in the example.

4.2 Correctness of ProgFromEx

We now present fundamental correctness properties of ProgFromEx. In particular, we define notions of soundness and completeness for inferring programs in TableProg, and claim that ProgFromEx is both sound and complete.

Theorem 1. *An table program inference algorithm is* sound *if whenever it infers a table program for an example input and output, then the program satisfies the examples. Formally, a table inference algorithm A is* sound *if for each input table T_I and output table T_O, if $A(T_I, T_O) = P$, then $P(T_I) = T_O$.* ProgFromEx *is sound.*

Proof. (sketch) Let P be as in Thm. 1. ProgFromEx is sound if and only if when P is applied to T_I, then (1) if P produces a cell, the cell is in T_O and (2) P produces all cells in T_O. Condition (1) holds, as P applies a coordinate map that is the union of maps implemented by its component programs. A component program is only added to P if the map that it implements is consistent with T_I and T_O, as discussed in §4.1. Condition (2) holds, as step 2) of ProgFromEx only returns a table program if ProgFromEx finds a set of component programs that map to all cells in the output table, or if ProgFromEx does not infer a table program. □

We now define completeness for a table inference algorithm, and argue that ProgFromEx is complete.

Theorem 2. *A table inference algorithm A is* defined *for an example input T_I and example output T_O only if when given the example, it infers a table program. A is* complete *for a language of table programs if whenever some table program P in the language satisfies T_I and T_O, then A is defined on T_I and T_O.* ProgFromEx *is complete for* TableProg.

Proof. (sketch) Let P be a table program that satisfies the condition on P in Defn. 2. We say that P can be built *subject to the elements* C, P, S, and R. Given C, P, S, and R, ProgFromEx infers a program P' that also satisfies the condition on P in Defn. 2. This follows from the fact that if C is a component program in P, then ProgFromEx finds a component program C′ that implements the same map as C. This follows by weak induction on the *size* of a component program, where the size of a filter program is 1, and the size of an associative program is one greater than the size of its component program. In particular, if a component program C built subject to the elements is of size n, then ProgFromEx finds a component program that implements the same map as C. This is proved as follows.

For the basis case, let C be a filter program built subject to the elements, with size $n = 1$. By hypothesis, $\mathsf{C} = \texttt{FILTER}(\mathsf{P}, \mathsf{S})$ implements a candidate map given by C. CondFromMap (Fig. 4, line [2]) finds all mapping conditions for a given set of candidate maps and predicates, so CondFromMap finds P. SeqOfMap (Fig. 4, line [3]) finds all output sequencers from a given set of output sequencers that can implement a given set of candidate maps. Thus SeqOfMap finds S. Thus in step 1), ProgFromEx finds C, and collects either C or another filter program that implements the same map as C.

For the inductive case, let $\mathsf{C} = \texttt{ASSOC}(\mathsf{C}_1, \mathsf{R}_1, \mathsf{R}_2)$ be an associative program built subject to the elements, with size $n > 1$. Assume as an inductive hypothesis that for each component program D of size $n-1$, ProgFromEx finds a component program that implements the map of D and can be built subject to the elements. C_1 is a component program of size $n-1$, so by the inductive hypothesis, ProgFromEx finds a component program $\mathsf{C}_1{}'$ that implements the same map as C. The associative program $\texttt{ASSOC}(\mathsf{C}_1{}', \mathsf{R}_1, \mathsf{R}_2)$ implements the same map as A. □

ProgFromEx is complete for inferring programs in TableProg. However, ProgFromEx cannot, in general, infer a table program for an arbitrary example input and output. Instead, TableProg and ProgFromEx can serve as a framework for table program languages and inference algorithms. To obtain a more expressive language of table programs, one can define more expressive sets of candidate-map rules, predicates, output sequencers, and relative functions. We have extended TableProg in this way in our implementation, and the resulting language is expressive enough to implement many transformations required by real users.

4.2.1 Correct for Examples vs. Correct for Expectations

ProgFromEx is effective for finding a table program that satisfies a given example. However, multiple programs in TableProg may satisfy an example, yet in general, implement different table transformations. Thus a user may provide to ProgFromEx an example, ProgFromEx may provide to the user a program that satisfies the example, and the user may then apply the inferred program to other tables only to find that the program does not behave as expected. Yet there may be a different program in TableProg that also satisfies the user's examples and also behaves as the user expects when it is applied to other inputs.

An inference algorithm can apply a variety of approaches to provide to the user the program that the user requires. First, an inference algorithm can *actively* query the user about the program they require until the algorithm finds a unique program that satisfies the user's requirements. This approach is analogous to the one described in [12], and has several nice properties. In particular, if the approach produces a program, then the program is unambiguously correct for all inputs. However, such an approach may need to query the user a prohibitively large number of times. We leave as future work the problem of developing an inference algorithm that follows this approach, yet queries a user a small number of times.

We instead developed ProgFromEx to apply a *lazy* approach. In a lazy approach, the inference algorithm takes an example from the user, and infers some program that satisfies the example. The user applies the inferred program to other inputs; if on another input, the program produces an output that the user does not expect, then the user provides the input and unexpected output to the inference algorithm as an example, and the algorithm infers a new program that satisfies both the original and new example. The user repeats this process until the inference algorithm provides a program that behaves as the user expects for the inputs on which they apply it. Unlike an approach based on active querying, the lazy approach does not guarantee that if the inference algorithm infers a program, then the program is correct for all inputs. However, we have observed that in practice, users do not need to apply table programs to arbitrary input tables. Instead, users apply a table program to a set of tables that all satisfy a strong condition. Requiring users to specify a program's behavior for tables that do not satisfy this condition is unnecessary, and often causes users to refuse to use such a technology. We now describe two extensions of ProgFromEx that allow users to apply it using a lazy approach to quickly infer the program that they require.

If a user applies ProgFromEx to a given example, obtains a program, and finds that the program behaves incorrectly on a different input, then the user can provide the second, different input, along with a corresponding correct output, as another example for ProgFromEx, and obtain a new program that better satisfies their requirements. If the second input extends the first input, then the user may apply ProgFromEx solely to the second input. However, even if the example inputs are incomparable, ProgFromEx can be extended to take multiple examples from a user simultaneously. To take multiple examples, ProgFromEx as presented in Fig. 4 is extended to find filter programs and associative programs that are consistent for a set of multiple examples. To find filter programs that are consistent for all examples, the loop at lines [1]–[6] is changed to enumerate over the space of all tuples containing a candidate map for each example. For each tuple of candidate maps, ProgFromEx attempts to infer a map condition that classifies exactly the cells mapped by each candidate map, and attempts to infer an output coordinate sequencer that describes the sequence of output cells that are mapped to in each candidate map. To find an associative program that is consistent for all examples, the check at line [14] is extended to determine if the associative program AssocProg is consistent with each example. Finally, the checks at lines [9] and [19] are extended so that ProgFromEx determines that it has found a satisfying table program only when it finds a collection of component programs that map to every cell in all output example tables.

ProgFromEx can also be extended so that it infers a program from a single example that is, in practice, more likely to behave as expected when applied to other tables. Step 2) of ProgFromEx halts when ProgFromEx finds a set of component programs that map to every cell in the example output table. However, the resulting set may include multiple component programs that are redundant, as a smaller set of programs would still map to the same cells. In practice, the more component programs that form a table program, the more likely the table program is to behave incorrectly when applied to other tables. This satisfies an informal notion of Occam's Razor: the simplest table program is often the best. We have thus extended ProgFromEx so that at Fig. 4 line [20], it does not necessarily build a table program from all component programs that it finds. Instead, ProgFromEx first applies a greedy algorithm to prune the set of all component programs found to a set that still maps to all cells in the example output, but is locally minimal. The resulting program is intuitively "simpler" than the original program, and in practice more often behaves as expected on larger examples.

4.3 Performance of ProgFromEx

We now analyze the complexity of ProgFromEx and present optimizations that allow it to perform significantly better in practice than its asymptotic worst-case.

4.3.1 Complexity of ProgFromEx

In the worst case, ProgFromEx may take time exponential in the size of the example tables given to it. Let n be the number of cells in the example input table, m be the number of cells in the example output table, p be the number of atomic predicates, s be the number of output sequencers, and r be the number of relative functions. We first observe that a map computed by a component program has at most n entries. This follows from two facts. First, the map computed by a filter program is a function, which follows directly from the semantics of a filter program. Second, an associative program computes its map by altering the map computed by its component program, but it does not add entries.

Recall from §4.1 that ProgFromEx executes in two steps: in step 1), ProgFromEx builds a set of filter programs from a set of

candidate maps. To infer a filter program from a candidate map, ProgFromEx first infers a map condition. To do so, ProgFromEx determines the values of all predicates in StatePreds as a hypothetical filter program reads each of the n input cells. This takes $O(np)$ time. ProgFromEx then intersects each set of $O(p)$, with one set for each of the n states; this takes $O(np)$ time. Thus for each candidate map, ProgFromEx infers a map condition in $O(np)$ time.

To infer a filter program, ProgFromEx infers an output sequencer in addition to inferring a mapping condition. To infer the sequencer, ProgFromEx examines each of the s possible sequencers, and checks if each sequencer is consistent for each of the $O(n)$ entries in the candidate map. Thus for a candidate map, ProgFromEx finds an output sequencer in $O(ns)$ time, and thus infers a filter program for a candidate map in $O(n(p+s))$ time. ProgFromEx only must infer filter programs for candidate maps that are functions over the n input cells. There are $O(m^n)$ such candidate maps, so ProgFromEx finds the set of all possible filter programs in step 1) in $O(m^n n(p+s))$ time.

In step 2), ProgFromEx infers a set of associative programs by iterating in a loop. Each time through the loop, ProgFromEx first checks the loop guard, which amounts to checking (1) if the set of new component programs is empty, and then checks (2) if the maps of all component programs that it has found cover the output table. Checking condition (1) takes constant time. Checking condition (2) can be optimized by changing the algorithm in Fig. 4 to maintain the set of all output coordinates that have not been mapped, and updating this set at line [15] when a new component program is found. Checking (2) then takes constant time, while requiring an $O(m)$ operation at line [15].

After checking the loop guard, ProgFromEx chooses a new component program CompProg from its worklist (Fig. 4, line [10]) and builds a set of r^2 new associative programs from CompProg. ProgFromEx checks if each new associative program is consistent for the example tables T_I and T_O by applying the program to T_I and checking if the associative program computes a map that produces a substructure of T_O. Each map has at most n entries, so ProgFromEx checks that the associative program is consistent in $O(n)$ time. ProgFromEx then checks if the map of the new associative program is distinct from the maps of all component programs found so far. If ProgFromEx maintains the set of all $O(m^n)$ such maps as an ordered list, then this check takes $O(\log m^n) = O(n \log m)$ time. ProgFromEx thus executes one iteration of the loop in the combined time of checking the loop guard, updating the set of unmapped coordinates, and building new associative programs, which is $O(1) + O(r^2(n \log m + m)) = O(mnr^2)$.

To analyze the complexity of step 2), we now need only bound the number of iterations of the loop. The number of iterations is bounded by the number of elements that ProgFromEx adds to Worklist, and ProgFromEx adds a new component program to Worklist only if the program's map is distinct from the maps of all other component programs that ProgFromEx has found so far. Each map has at most n entries, which implies that $O(m^n)$ maps are added to the worklist. Thus ProgFromEx completes step 2) in $O(m^{n+1}nr^2)$ time. Finally, after step 2), ProgFromEx checks if it has inferred a set of component programs whose maps cover the entire output table, which can be done in constant time. Thus ProgFromEx executes in total time $O(m^{n+1}n(p+s+r^2))$.

4.3.2 Performance in Practice

If ProgFromEx performed close to its worst-case bound, then it would be highly impractical. Fortunately, real-world input and output examples have properties that allow ProgFromEx to execute quickly, or that allow for heuristics that greatly improve its performance. In particular, the dominating factor in the high complexity of ProgFromEx is the set of component programs with distinct maps that ProgFromEx may collect. In practice, this set is quite small. This is because while many component programs may implement a large set of distinct maps according to the combinatorial bound, in practice only a small set of these maps are consistent for the example tables. Thus candidate-map rules find few candidate maps that are consistent with the examples, and when ProgFromEx checks if an associative program is consistent (Fig. 4 line [14]), the check rules out many potential associative programs immediately.

ProgFromEx as given in Fig. 4 can also be optimized by biasing the order in which it picks component programs to build new associative programs. In line [10] of Fig. 4, ProgFromEx nondeterministically chooses a component program from its worklist, and builds new associative programs from the chosen component program. The chosen component program may mostly map to output cells that are already mapped to by other component programs in Comps, as may the associative programs built from the chosen component program. However, ProgFromEx can instead prioritize the elements in its worklist so that ProgFromEx first chooses component programs from the worklist that map to many cells not mapped to by any component program in Comps. This may allow ProgFromEx to more quickly find a set of component programs that cover the example output table.

5. Experiments

We implemented an interpreter for TableProg, implemented ProgFromEx, and experimented with the implementations to determine if TableProg and ProgFromEx are useful in practice. The experiments were designed to determine the following:

1. Is TableProg expressive enough to describe table transformations that real end-users require?
2. When a program in TableProg implements the table transformation that a user requires, could ProgFromEx find the program quickly?
3. If ProgFromEx finds a program that satisfies a user's initial examples, and the program is applied to other similar inputs, does the program produce the expected outputs? If not, how many additional examples does a user need to provide before ProgFromEx infers a program that behaves as expected?

We performed the experiments as follows. We searched two online help forums[2] for the Excel spreadsheet program to find table transformations that real users needed to perform, but could not accomplish using the built-in GUI features of Excel. Among the transformations, we picked 51 for which the user provided an example input and output. We gave the input and output examples to ProgFromEx, which inferred a table program for each example. We used the provided examples and the English description of the transformation to manually create a larger input example, applied the inferred program to the larger input, and checked that the program produced the output that we expected. If the program did not produce the expected output, then from the unexpected output, we manually derived a more descriptive example, and applied ProgFromEx to the more descriptive example. We repeated this process until ProgFromEx found a program that behaved as expected for successively larger, different inputs.

The experiments indicate that TableProg describes practical programs, and that ProgFromEx finds practical programs in TableProg efficiently. The times spent by ProgFromEx to infer programs for the 51 tests are presented in Fig. 5. For each of the 51 tests, ProgFromEx found in less than 10 seconds a program in TableProg that satisfied the example. For a majority (31) of the

[2] http://www.excelforum.com and http://www.ozgrid.com/forum/

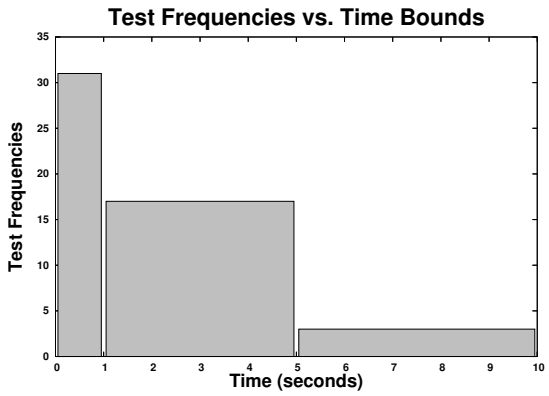

Figure 5. Time for ProgFromEx to infer table transformation programs.

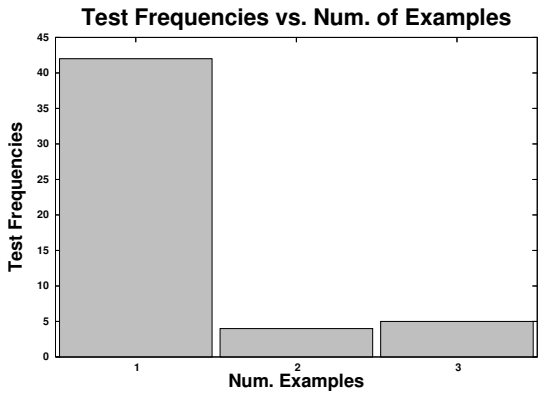

Figure 6. Number of examples required by ProgFromEx to infer programs.

tests, ProgFromEx found a program in less than a second. For a vast majority (48) of the tests, ProgFromEx found a program in less than 5 seconds.

If for a given test, ProgFromEx inferred a program that did not behave as we expected when applied to other inputs, we manually found an input on which ProgFromEx misbehaved and reapplied ProgFromEx to the new input. We repeated this process until ProgFromEx inferred a program that behaved as expected. The numbers of times that we needed to refine examples for the tests are presented in Fig. 6. Observe that for a vast majority (42) of the tests, ProgFromEx inferred the expected program from one example. For the four of the tests, ProgFromEx required two examples, and for five of the tests, it required three examples.

The large number of different help-thread requests that were satisfied by programs in TableProg indicates that TableProg is expressive enough to describe transformations required by real end-users. Moreover, the transformations requested on the help threads are diverse. To illustrate their diversity, we discuss a selection of examples that are satisfied by programs in TableProg. The examples are slightly simplified from their original forms to ease presentation. For a complete list of help threads containing the

Example input table:

55660	x1		$530.00		
55660	x3	11/5/2007	$10.00	5	$2,130.00
90210	y1		$25.00		
90210	y2		$25.00		
90210	y3	11/18/2007	$25.00	13	$325.00

Example output table:

55660	11/5/2007	$10.00	$2,130.00
90210	11/18/2007	$25.00	$325.00

Figure 7. Example reproduced from an Excel help thread "Copy Rows Where Column Cell Is Not Blank"

Example input table:

Name	Colour	Price
Toyota	White	2000
Nissan	Red	4000

Example output table:

Toyota	White
Toyota	2000
Nissan	Red
Nissan	4000

Figure 8. Example reproduced from Excel help thread "arranging data using VBA."

original examples on which we experimented, see [6]. For detailed descriptions of the tests, see App. A.

Filtering Key Information In many help requests, users need to transform a table to retain a subset of the table's original information. This subset is defined by key cells in the table, along with cells that are spatially related to the key cell. In a help thread titled "Copy Rows Where Column Cell Is Not Blank," a novice user specified a table transformation using an example reproduced in Fig. 7. The user wanted to transform their input table into a new table that only contains the cells in columns 1, 3, 4, and 6 of each *summary row* of the input table. According to the user's English description, a summary row is any row in which column 6 is not empty. An expert user replied that a GUI feature partially automates this transformation, but the novice replied that they would like to fully automate the transformation. The expert replied with an Excel macro, which presumably satisfied the novice. When we applied ProgFromEx to the example given by the novice, ProgFromEx automatically found a TableProg program that satisfies the example in 0.5 seconds. However, when we reapplied the program to other input tables, we found that it placed cells from a row of the input table in the output table if and only if column 5 was empty. To resolve this, we extended the input to have a row in which the column 5 was empty but column 6 was non-empty, and extended the output to contain cells from the row. Using this example, ProgFromEx inferred a program, again in 0.5 seconds, that behaved as expected when reapplied to other tables. The program consists of a filter program which maps the non-empty cells in column 6 of the input to column 4 of the output. For each of the other three columns in the output, an associative program built from the filter program maps the column.

Splitting and Partially Replicating Rows In many help requests, users need to split cells in a row into different rows, while replicating other cells in the original row. In a help forum post titled "Arranging Data Using VBA," a novice user requested a table transformation by providing only the example reproduced in Fig. 8. The user added that though they had provided a small example, they needed to apply a transformation to a table that had thousands of rows. After more than four hours, an expert user provided an Excel macro that presumably satisfied the novice's request. We applied ProgFromEx to the example from the help forum, and ProgFromEx automatically found in 2.3 seconds a table program that satisfies the example. When we reapplied the program to larger, similar examples, it behaved as expected. The table program con-

Example input table:

3099	905	A4CA				
NO.14	NO.14	Full Copies	6.78	2	* *	0
3200	906	AHG				
9-Jun	9-Jun	Covers Only	4.74	1	* *	0

Example output table:

3099	905	A4CA	NO.14	Full Copies	6.78	2
3200	906	AHG	9-Jun	Covers Only	4.74	1

Figure 9. Example reproduced from an Excel help thread "arrange data."

Example input table:

Alice	Art&Des	B
	CreatArt	A
	D&T	A
	English	A
	Geo.	A*
Bob	Art&Des	C
	CreatArt	B
	D&T	C
	English	C
	Geo.	C

Example output table:

	Art&Des	CreatArt	D&T	English	Geo.
Alice	B	A	A	A	A*
Bob	C	B	C	C	C

Figure 10. Example reproduced from an Excel help thread "Transposing 3 columns into multiple columns."

sists of a filter program that maps cells in columns 2 and 3 of the input to column 2 of the output, and a single associative program built from the filter program that maps to column 1 of the output.

Combining and Filtering Rows In many requests, users need to remove some cells from rows while combining adjacent rows. In a help thread titled "arrange data," a novice user requested a table transformation by providing only the example reproduced in Fig. 9. The transformation concatenates adjacent pairs of rows, filtering out the cell in column 1 in the second row of each pair. In twenty minutes, an expert user provided to the novice a macro that satisfied the example, and the novice reported that they were satisfied, although they did not understand the mechanics of the macro. When we applied ProgFromEx to the example in Fig. 9, it automatically found in 1.2 seconds a TableProg program that implements the transformation requested by the user. When we reapplied the same program to larger, similar examples, it behaved as expected. The program is built from two filter programs that map onto columns 6 and 7 of the output table, two associative programs built from the filter programs that map onto columns 3 and 4 of the output table, a third associative program, built from the associative program that maps onto column 4, that maps onto column 2, and finally a fourth associative program built from the third associative program, that maps onto column 1. This example demonstrates the benefit to the iterative approach applied by ProgFromEx to find associative programs.

Rearranging Groups In many requests, a user has data organized in nested groups, and needs to preserve the grouping of the data while rearranging the layout of the groups. In a help thread titled "Transposing 3 columns into multiple columns," a novice user requested a transformation by providing the example reproduced in Fig. 10. In the novice's input table, grades are arranged vertically, grouped by student and then by subject. In the novice's output table, grades are arranged horizontally, grouped in a row per student, and a column per subject. After the novice posted their request, one

Example input table:

PROJ	CAT	SPONSOR	DEPT	ELTS	DUE
SPEC	OOH	Infiniti	Design	elt 1	11/10
SPEC	OOH	Infiniti	Desing	elt 2	
SPEC	Print		Design	elt 3	11/30
SPEC	Print		Design	elt 4	11/30
SPEC	Print	Infiniti	Design	elt 5	11/30

Example output table:

		SPEC	
		OOH	
Infiniti	Design	elt 1	11/10
Infiniti	Desing	elt 2	
		Print	
	Design	elt 3	11/30
	Design	elt 4	11/30
Infiniti	Design	elt 5	11/30

Figure 11. Example reproduced from Excel help thread "printed sheet in different format than worksheet."

expert user replied with a macro about 40 minutes later, and another expert replied with a macro nearly six hours after the request was posted.

When we applied ProgFromEx to the example in Fig. 10, it automatically found a TableProg program in 1.4 seconds that implements the requested transformation. When we reapplied the same program to larger, similar tables, the program behaved as expected. The table program includes a single filter program that maps to all cells at or below row 2 between columns 2 and 5. One associative program built from this filter program maps onto all cells in column 1 of the output table. A second associative program built from the filter program maps onto all cells in row 1 of the output table.

In a help thread titled "printed sheet in different format than worksheet," a novice user provided the example reproduced in Fig. 11. In the novice's input table, each row of the table is a complete entry for an element of a project. The novice needed to group the elements so that the project and category (columns 1 and 2, respectively) of the element appear as headings and subheadings, followed by the elements that are under the heading and subheading. Although the novice suspected that elements could be grouped semi-automatically using built-in features of Excel, they did not know how to do so. No expert user helped the novice with their request.

When we applied ProgFromEx to the example in Fig. 11, it automatically found a TableProg program in 4.4 seconds that implements the required transformation. In the TableProg program, a filter program maps to column 3 of the output. A set of three associative programs built from the filter program map to columns 1, 2, and 4, with one associative program mapping to each column.

We implemented the interpreter for TableProg and the inference algorithm ProgFromEx as an Excel plug-in module. To use the module, a user selects in the Excel GUI a range of cells to act as the example input, selects a range of cells to act as the example output, selects a range of cells to act as a "full input," selects a location in the spreadsheet where they would like to place the output corresponding to the full input, and pushes a button. The module then automatically infers a program that satisfies the examples, applies the program to the full input, and places the resulting output at the location specified by the user. We plan to extend the interface so that a user may supply only examples, and the module will provide a program that the user can add to a library. The user could then reapply programs in the library whenever they wish.

6. Related Work

The area of program synthesis is gaining renewed interest [8], and this paper is a work in this direction. Program synthesis has traditionally been motivated by the need to synthesize non-trivial algorithms [11, 21, 22] or to discover tricky code-snippets [10, 23]. In this paper, we apply program synthesis to discovering relatively simpler programs, but those that are cared about by a much larger class of spreadsheet end-users, who often struggle with manipulating tabular data in spreadsheets.

There has been some work in the programming languages and the HCI community related to inferring table transformations. PADS [7] takes a large sample of unstructured data and infers a format that describes the data. Users then manually define tools for data in the format. Lenses [2] are a language of combinators that can be applied to solve the *view update problem*, which is related to but distinct from the problem of inferring a transformation over tables from examples. The Wrangler tool, developed in the HCI community, provides a nice visual programming-by-demonstration interface to table transformations for data cleaning [13]. In contrast, we provide an interface based on examples, which is more friendly to end users.

Program sketching [20] may be applied to synthesize programs from examples. In program sketching, a programmer supplies to a sketcher an incomplete version of a program (a *sketch*) and a specification of correct behavior for the program, possibly as a set of input-output examples. The sketcher then completes the program so that the program satisfies the specification. We attempted to apply the SKETCH [19] program sketcher to a sketch of a table program and example tables. However, because SKETCH is a tool for synthesizing general programs, it could not take full advantage of the particular structure of table programs. As a result, when we applied it to infer programs in TableProg, it typically timed out after an hour.

The problem of inferring transformations over tables is related to inferring queries over relations from instances of the relations. These techniques cannot be directly applied to infer transformations over tables, as users treat tables differently from relations; e.g., tables have a notion of order over rows that does not hold by default for relations. In particular, in the *view synthesis problem* [4], one takes a database relation and a view over the database and infers the general query that produces the view. The technique presented in [4] only considers views that are a subset of the original database. Thus the technique cannot be applied to any of the examples given in this paper, or many of the other real-world cases that we studied. The problem of *query by output* [24] is similar to the view synthesis problem, but the technique presented in [24] infers select-project-join queries. Of the examples presented in this paper, such queries can only be applied to satisfy the example in Fig. 7, and such queries cannot be applied to many of the other real-world cases that we studied. We leave as future work the problem of applying our techniques to learn richer queries from instances of databases and views.

The work in [9, 14, 15, 17, 25] gives techniques that take textual input-output examples and infer programs that transform text. However, these techniques only infer transformations over plain text, or strings. Thus they cannot be directly applied to infer transformations over tables. While the techniques can, in principle, be applied by representing tables in plain text, it is then difficult to reason about the spatial relationships between cells in the original table. TableProg represents spatial relationships directly, and the relationships are critical to TableProg's ability to express practical table transformations. Furthermore, the techniques for textual transformations assume that it is easy to find a set of programs that satisfy an individual example of a transformation, and focus on combining sets of programs that satisfy different examples. However, when searching for programs that transform tables, it is non-trivial in general to find a set of programs that satisfy even a single example of a transformation.

7. Conclusion

End users often need to transform semi-structured, tabular data in non-trivial ways. We have presented a language TableProg of programs that implement table transformations that real users require, and an algorithm ProgFromEx that from an example input and output, infers a TableProg program that implements the transformation specified by the example. To demonstrate that TableProg and ProgFromEx are practical, we applied them to infer programs for over 50 table transformations specified as input-output examples by real end users.

A. Help Forum Test Cases

In this section, we describe the layout problems in Excel help forums on which we evaluated ProgFromEx. For each problem, we give the title of the help forum post, an `http` link to the help forum post, a description of the format of the input table, and a description of the output format of the table. We discuss the strings in the cells of the table simply as syntactic strings, and do not consider their semantic meaning to the user. For a semantic discussion, see the original help threads in which the examples are posted.

adding values from different cells (http://www.excelforum.com/excel-programming/731450-adding-values-from-different-cells.html)

Input table: columns A-L are non-empty. If two adjacent rows have identical values in any column other than G, then they have identical values in all columns other than G. The values in G are integers.

Output table: each group of rows in the input table with identical values in non-G columns is replaced with a single row with the same value in all non-G columns, and the sum of all G rows in the original group in the G column.

arrange data (http://www.excelforum.com/excel-new-users/688736-arrange-data.html)

Input table: different formats for alternating rows. Each odd-numbered row has columns A-C non-empty, and each even-numbered row has columns A-G non-empty.

Output table: each row is columns A-C of each odd-numbered row in the input concatenated with columns B-G of the following even-numbered row in the input.

Arrange data (http://www.excelforum.com/excel-general/718408-arrange-data.html)

Input table: a sequence of groups of rows. The first two rows in each group are non-empty in exactly column A. All following rows in the group are non-empty in columns A-B.

Output table: a sequence of identical groups, except the first row in each group is empty, and the value previously in the first row, column A is the second row, column B.

arranging data using vba (http://www.excelforum.com/excel-programming/635849-arranging-data-using-vba.html)

Input table: each row is non-empty in exactly columns A - C.

Output table: for each row in the input, a row with the same values in exactly columns A - B, followed by a row with the same value in column A and the value in column C of the input in column B of the output.

columns datas to row and insert new rows (http://www.excelforum.com/excel-programming/678030-columns-datas-to-row-and-insert-new-rows.html)

Input table: a sequence of groups of rows. All rows are non-empty in exactly cells A - H. Each group has the same value in column A.

Output table: a sequence of groups of rows corresponding to groups in the input table. All rows are non-empty in exactly cells A - B. Each group has the same value in column A, which is the same as the value in column A for the corresponding group. The values in column B are the values of all cells in columns B - H in the input group, in row-major order.

convert a table into a list (http://www.excelforum.com/excel-programming/659039-convert-a-table-into-a-list.html)

Input table: a header row of dates in columns B - D, then a blank row, then a sequence of rows that are non-empty in columns A - D.

Output table: a sequence of rows that are non-empty in columns A - C. Each row corresponds to a cell in column B - D in the input table. In the output row, column A is a the date at the head of the input cell, column B is the value in column A of the same row as the input cell, and column C is the value in the input cell.

Convert Data Into Reportable Format (http://www.ozgrid.com/forum/showthread.php?t=98196)

Input table: a sequence of groups. Each group is a sequence of rows with non-empty columns A - E, and the same value in column A.

Output table: each row corresponds to a group in the input. Column A of the output has the same value as column A of the group. The following columns in the output row are the sequence of values in column E of the input group.

Convert Detailed Table Into Summary Table (http://www.ozgrid.com/forum/showthread.php?t=141261)

Input table: each row has columns A - O non-empty. The header row contains titles, with the numbers 1 - 12 in columns D - O. Each following row is non-empty in columns A - C, and has an integer or blank in columns D - O.

Output table: for each input cell in columns D - O, an output row. Columns A - C of the output row have the same value as columns A - C in the same row as the input cell. Column D of the output row contains the column header of the input cell. Column E contains the value of the input cell.

convert multiple columsn to rows (http://www.excelforum.com/excel-worksheet-functions/333764-convert-multiple-columns-to-rows.html)

Input table: each row is non-empty in columns A - D.

Output table: for each row in the input, a group. Column A of the first row of the group is column A of the input row. Column A of the second row of the group is column B of the input row. Column A of the third row of the group is column C of the input row. Column A of the fourth row of the group is column D of the input row.

converting columns to rows using a loop (http://www.excelforum.com/excel-programming/694544-converting-columns-to-rows-using-a-loop.html)

Input table: each row is non-empty in columns A - F.

Output table: for each cell in columns B - F of the input, a row. Column A of the output row is column A of the row of the input cell. Column B of the output row is value of the input cell.

converting horizontal data to vertical data with multiple rows (http://www.excelforum.com/excel-general/698622-converting-horizontal-data-to-vertical-data-with-multilple-rows.html)

Input table: a header row for columns A - F. Each following row is non-empty in column A, and contains an optional entry in columns B - F.

Output table: for each non-empty cell in columns B - F of the input, a row. Column A of the row is column A of row of the input cell. Column B of the row is the column header of the input cell. Column C of the row the input cell.

Copy Rows Where Column Cell Is Not Blank (http://www.ozgrid.com/forum/showthread.php?t=80736)

Input table: each row may have non-empty values in columns A - I.

Output table: each row in the input table that is non-empty in column A.

Copy & Transpose Table Data Into Database Format (http://www.ozgrid.com/forum/showthread.php?t=140756)

Input table: a header row for columns A - G. Each following row is non-empty in columns A - B, and has an optional value in column C - G.

Output table: for each non-empty cell in columns C - G of the input, an output row. Column A of the output row is column A of the row of the input cell. Column B of the output row is column B of the row of the input cell. Column C of the output row is column header of the input cell. Column D of the output row is the input cell.

copy column without repeated number
(http://www.excelforum.com/excel-general/631382-copy-column-without-repeated-number.html)

Input table: a sequence of groups of rows. All rows are non-empty in columns A - C, and all entries in column B are integers. All rows in a group have the same value in column C.

Output table: for each group of rows in the input, a row in the output. Column A of the output row is the sum of all entries in column B of the group. Column B of the output row cell in column C of all rows in the group.

Copy Tranpose & Reformat Based on Changes in Value
(http://www.ozgrid.com/forum/showthread.php?t=95119)

Input table: a sequence of groups of rows. All rows are non-empty in columns A - B and F. All rows in the group have the same value in column A.

Output table: for each group in the input table, a row in the output table. Column A of the output row is column A of the group. Each following cell in the output row is column F in a row of the group.

Create report to show totals for vendors per month
(http://www.excelforum.com/excel-general/710914-create-report-to-show-totals-for-vendors-per-month.html)

Input table: each row is non-empty in columns A - D. Each cell in column D is a real.

Output table: in the output first row, column A is empty, and for each value in column A of some row of the input table, there is column in the output row with the value. For each value in column C of some row in input table, there is a row in output table with the same value in column A. Each value in a non-A column of any row other than the first row in the output is the sum of all values in column D of some input row that contain the output column header in input column A and output row header in input column C.

creating basic macros in excel (http://www.excelforum.com/excel-general/371934-creating-basic-macros-in-excel.html)

Input table: a sequence of rows, non-empty in columns A - J.

Output table: for each row of the input, a row of the output. Column A of the output row is column E of the input row. Column B of the output row is column B of the input row.

data arrangment
(http://www.excelforum.com/excel-programming/ 580849-data-arrangement.html)

Input table: a sequence of groups of rows. All rows are non-empty in columns A - G. All rows in a group have the same value in columns A - C.

Output table: for each input group, an output row. Columns A - C of the output row are columns A - C of the input group.

Data From Multiple Rows Into Single Column
(http://www.ozgrid.com/forum/showthread.php?t=78553)

Input table: all rows are non-empty in columns A - D.

Output table: a row for each cell in the input table, with column A of the output row the same as the input cell. The cells are printed in row-major order.

date grouping (http://www.excelforum.com/excel-general/736780-date-grouping.html)

Input table: a sequence of groups. All rows are non-empty in columns A - D. All cells in column D hold integer values. All rows in a group have the same value in columns A - C.

Output table: for each group in the input table, a row. Columns A - C of the output row are columns A - C of the input group. Column D of the output row is the sum of all cells in column D of rows in the group.

dectecting changes in excel (http://www.excelforum.com/excel-general/720989-dectecting-changes-in-excel.html)

Input table: a sequence of groups of rows. All rows are non-empty in column A.

Output table: Column A of each row is equal to column A of the corresponding input row. Column B is "J" if the value in column A is not equal to the value of column A in the previous row. Otherwise, column B is empty.

excel data manipulation kinda like a transpose
(http://www.excelforum.com/excel-general/737216-excel-data-manipulation-kinda-like-a-transpose.html)

Input table: a sequence of groups. All rows are non-empty in columns A - B. All rows in a group have the same value in column A.

Output table: for each group in the input, a row in the output. Column A of the output row is column A of the group. The values in column B of the rows of the input group are then concatenated in the columns of the output row.

how to rearrange rows in excel (http://www.excelforum.com/excel-general/733195-how-to-rearrange-rows-in-excel.html)

Input table: a sequence of groups. All rows are non-empty in columns A - B. All cells in column B contain "A", "B", or "C." All groups have the same value in column A.

Output table: a header row where columns B - D are "A, B, C". Then, for each group in the input table, a row in the output table. Column A of the output row is column A of the group. The column headed by "A" is "Yes" if there is a row in the group with "A" in column B, and "No" otherwise. Similarly for the columns headed by "B" and "C."

how to transpose variable data from column a to multiple rows
(http://www.excelforum.com/excel-programming/739908-how-to-transpose-variable-data-from-column-a-to-multiple-rows.html)

Input table: a sequence of groups. Each group is a sequence of rows with only column "A" non-empty. Each group is separated by a blank row.

Output table: a row for each group in the input table. Each column in the row is column A of a row in input group.

importing data into excel spreadsheet (http://www.excelforum.com/excel-general/641950-importing-data-into-excel-spreadsheet.html)

Input table: a sequence of rows where column A is non-empty.

Output table: a single row where each column is column A of a row in the input. The column entries in left-right order are the rows of the input in descending order.

more than a simple column to row transformation (http://www.excelforum.com/excel-general/729574-more-than-a-simple-column-to-row-transformation.html)

Input table: a sequence of groups. In each row, only column A is non-empty. Each group contains exactly three rows.

Output table: for each input group, a row. Each row is non-empty in columns A - D. Column A of the output row is column A of the first row of the group, column B is column A of the second row of the group, column C is column A of the third row of the group, and column D is column A of the fourth row of the group.

Move Single Row Ranges Of Column to 1 Row (http://www.ozgrid.com/forum/showthread.php?t=107947)

Input table: a sequence of rows, where each row is non-empty in columns A - B.

Output table: a single row, which is the concatenation of all rows in the input.

need some sort of macro to arrange file (http://www.excelforum.com/excel-programming/349139-need-some-sort-of-macro-to-arrange-file.html)

Input table: a sequence of groups. Each row of a group is non-empty in columns A - B. Each group contains three rows. Each group is separated by a row of empty cells.

Output table: for a input group, a pair of rows, each non-empty in columns A - C. For each odd-numbered output row, column A is column A of the first row of the input group, column B is column A of the second row of the input group, and column C is column A of the third row of the group. For each even-numbered output row, column A is column B of the first row of the input group, column B is column B of the second row of the input group, and column C is column B of the third row of the group.

pivot tables simple question (http://www.excelforum.com/excel-general/349735-pivot-tables-simple-question.html)

Input table: a sequence of rows. Each odd-numbered row is non-empty in columns A - F, and each even-numbered row is non-empty in columns B - F.

Output table: for each pair of input rows in the input table, a single row in the output table. Column A of the output row is column A of the first row in the pair. Columns B - E of the output row are columns C - F of the first row in the pair. Columns F - I of the output row are columns C - F of the second input row in the pair.

Re-arrange Table & Transpose Headings (http://www.ozgrid.com/forum/showthread.php?t=85680)

Input table: a sequence of groups. Each row in non-empty in columns A - D. Each row in a group has the same value in column A.

Output table: for each input group, an output row. Column A of the output row is column A of the input group. For each row in the group, there is a column in the output row with column D of the input row.

Re-arrange 1 Column To Rows Based On Criteria

Input table: a sequence of groups. Each row is non-empty in columns A - B. Each group has the same value in column A.

Output table: for each group, a row. Column A of the output row is column A of the group. Then, for each row of the input group, the output row has a column with column B of the input row.

printed sheet in different format than worksheet (http://www.excelforum.com/excel-miscellaneous/711231-printed-sheet-in-different-format-than-worksheet.html)

Input table: a sequence of groups. Each group is a sequence of subgroups. All rows are non-empty in columns A - F. All rows in the same group are equal in column A. All rows in the same subgroup are equal in columns A - B.

Output table: For each input group of rows, an output group of rows. The first row in the output group is has column C equal to column A of the group. For each subgroup in the group, the group contains a row in which column C is column of the subgroup, and then for each row in the input subgroup, an output row, where columns A - D of the output rows are columns B - E of the input row.

re-arranging cells (http://www.excelforum.com/excel-general/349770-re-arranging-cells.html)

Input table: a sequence of rows. Each row is non-empty in column A.

Output table: a single row. Each column is equal to column A of an input row. The rows of column A in descending order are equal to the columns in left-to-right order.

Reformat Table Layout (http://www.ozgrid.com/forum/showthread.php?t=141260)

Input table: a sequence of groups. Each row is non-empty in columns A - F. Each group is equal in columns A - B.

Output table: for each group in the input table, an output row. Columns A - B of the output row are columns A - B of the group. The next columns of the output row are then columns C - F of each row in the group.

removing unwanted rows and summing the duplicate rows (http://www.excelforum.com/excel-programming/732253-removing-unwanted-rows-and-summing-the-duplicate-rows.html)

Input table: a sequence of groups. Each row is non-empty in columns A - N. All cells in column C - N are integers. All rows in the group are in equal in columns A - B.

Output table: for each group, a row. Columns A - B of the output row are columns A - B of the group. Column C is the sum of all column C cells in all rows of the group, and similarly for cells D - N.

solved rotate tranpose data (http://www.excelforum.com/excel-general/719400-solved-rotate-tranpose-data.html)

Input table: a sequence of groups. The first row of each group is non-empty in columns A - C, and all following rows in each group are non-empty in columns B - C.

Output table: for each group, a row. Column A of the row is column A of the group. The next columns are column C of the rows of the group.

Split & Reformat Data Layout (http://www.ozgrid.com/forum/showthread.php?t=79593)

Input table: a sequence of groups. Each row is non-empty in column A. Each group is 11 rows.

Output table: for each group, a row. Columns A - K of each row are the cells of column A of each row in the group, in descending order.

subtotals in a sheet (http://www.excelforum.com/excel-programming/580722-subtotals-in-a-sheet.html)

Input table: a sequence of groups. Each row is non-empty in columns A - E. Columns C - E are integer. Each row in a group is equal in column A.

Output table: for each group, a row. Column A of the output row is column A of the input group. Column B of the output row is the sum of all cells in column C of rows in the group, column C of the output row is the sum of all cells in column D of rows in the group, and column E of the output row is the sum of all cells in column F of rows in the group.

transform column to row (http://www.excelforum.com/excel-2007-help/737891-transform-column-to-row.html)

Input table: a sequence of groups. Each group is two rows non-empty in column A. Groups are separated by a single empty row.

Output table: a pair of rows. The columns of the first row are the cells in column A of the first row of each group. The columns of the second row are the cells in column B of the second row of each group.

transpose column to row (http://www.excelforum.com/excel-general/736049-transpose-column-to-row.html)

Input table: a sequence of rows. Each row is non-empty in columns A - D.

Output table: a single row. The columns of the row are the concatenation of all rows in the input table.

Transpose Data
(http://www.ozgrid.com/forum/showthread.php?t=144668)

Input table: a sequence of rows. Each row is non-empty in columns A - E, and is then non-empty for sequence of columns that is a non-zero multiple of four.

Output table: for each row, a group. The first row of each group is non-empty in columns A - I. Columns A - F of the first row are columns A - D of the input row, and columns E - I are the next four columns of the input row. Each next row of the group is empty in columns A - F, and columns G - J are the next four non-empty columns of the input row.

Transpose data in Access
(http://www.ozgrid.com/forum/showthread.php?t=144832)

Input table: a sequence of groups. Each row of the group is non-empty in columns A - E. All rows in the group are equal in column B.

Output table: for each input group, an output row. Column A of the output row is column B of the group. The next columns are columns C - E of each row in the group.

Transpose Groups of Cells From Columns To Rows
(http://www.ozgrid.com/forum/showthread.php?t=135846)

Input table: a sequence of groups. Each row is non-empty in column A. Each group is a sequence of four rows.

Output table: for each group, a row. Column A of the output row is column A of the first row of the input group, column B of the output row is column A of the second row of the input group, column C of the output row is column A of the third row of the input group, and column D of the output row is column A of the fourth row of the input group.

Transpose Groups of Data Seperated by Blank Rows
(http://www.ozgrid.com/forum/showthread.php?t=143431)

Input table: a sequence of groups. Each row in a group is non-empty in columns A - B. Each group contains five rows. Each group is separated by a blank row.

Output table: for each input group, an output row. Column A of the output row is column B of the first row in the group. Column B of the output row is column B of the second row in the group. Column C of the output row is column B of the third row in the group. Column D of the output row is column B of the fourth row in the group.

Transpose Data Conditionally
(http://www.ozgrid.com/forum/showthread.php?t=98089)

Input table: a sequence of groups. Each row is non-empty in columns A - B. All rows in the group are equal in column A.

Output table: for each input group, an output row. Column A of the output row is column A of the input group. The next columns of the output row are the cells in column B of the rows in the input group.

Transpose Multiple Columns Into Rows Based On Repeats
(http://www.ozgrid.com/forum/showthread.php?t=138959)

Input table: a sequence of groups. Each row is non-empty in columns A - C. All rows in a group are equal in column A.

Output table: for each input group, an output row. Column A of the output row is column A of the input group. The next columns of the output row are columns B - C of each row in the input group.

tranpose range into table (http://www.excelforum.com/excel-general/737865-transpose-range-into-table.html)

Input table: a sequence of groups of rows. Each row is non-empty in column A. There are five rows in each group.

Output table: for each input group, an output row. Column A of the output row is column A of the first row in the input group, column B is column A of the second row in the group, column C is column A of the third row in the group, column D is column A of the fourth row in the group, and column E is column A of the fourth row in the group.

Transpose Single Column Data to Multiple (Variable) Rows
(http://www.ozgrid.com/forum/showthread.php?t=145049)

Input table: a sequence of groups. Each row in a group is non-empty in column A. Each group is separated by a non-empty row.

Output table: for each input group, an output row. Each column of the output row is column A of a line in the input group.

transposing 3 columns into multiple columns
(http://www.excelforum.com/excel-2007-help/737427-transposing-3-columns-into-multiple-columns.html)

Input table: a sequence of groups, each containing 12 rows. The first row of each group is non-empty in columns A - C, and all following rows in the group are non-empty in columns B - C.

Output table: for each input group, an output row. Column A of the output row is column A of the first row of the group. The next columns of the output row are then the cells in column C of rows of the group.

turn 1 column into 2 first cell col 1 2nd col b 3rd delet...

Input table: a sequence of groups. Each group is a pair of rows. Each row in the group is non-empty in column A. Groups are separated by an empty row.

Output table: For each group, a row. Column A of each output row is column A of the first row of the group, and column B of the output row is column A of the second row of the group.

Using a macro to extract and rearrange data?
(http://www.excelforum.com/excel-programming/698490-using-a-macro-to-extract-and-rearrange-data.html)

Input table: a header row that is empty in column A, and non-empty in column B - D. Every other input row is non-empty in columns A - D.

Output table: for each non-empty cell not in the input header row and not in column A, an output row. Column A of the output row is the row header of the row containing the input cell. Column B of the output row is the column header of the column containing the input cell. Column C is equal to the input cell.

References

[1] R. Abraham and M. Erwig. Header and unit inference for spreadsheets through spatial analyses. In *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC)*, pages 165–172, 2004.

[2] D. M. Barbosa, J. Cretin, N. Foster, M. Greenberg, and B. C. Pierce. Matching lenses: alignment and view update. In *ICFP '10*, 2010.

[3] A. Cypher, editor. *Watch What I Do – Programming by Demonstration*. MIT Press, Cambridge, MA, USA, 1993. Full text available at web.media.mit.edu/ lieber/PBE/.

[4] A. Das Sarma, A. Parameswaran, H. Garcia-Molina, and J. Widom. Synthesizing view definitions from data. In *ICDT '10*, 2010.

[5] Excel. Microsoft Excel, 2010. URL http://office.microsoft.com/en-us/excel.

[6] Excel Help Forums. Excel help forum threads, 2010. URL http://cs.wisc.edu/∼wrharris/pldi2011/tests.html.

[7] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: fully automatic tool generation from ad hoc data. In *POPL '08*, 2008.

[8] S. Gulwani. Dimensions in program synthesis (invited talk paper). In *ACM Symposium on PPDP*, 2010.

[9] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330, 2011.

[10] S. Gulwani, S. K. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[11] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. A simple inductive synthesis methodology and its applications. In *OOPSLA*, pages 36–46, 2010.

[12] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE '10*, 2010.

[13] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *ACM Human Factors in Computing Systems (CHI)*, 2011.

[14] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. Programming by demonstration using version space algebra. *Mach. Learn.*, 53(1-2): 111–156, 2003. ISSN 0885-6125.

[15] T. A. Lau, P. Domingos, and D. S. Weld. Version space algebra and its application to programming by demonstration. In *ICML '00*, 2000.

[16] H. Lieberman, editor. *Your wish is my command: programming by example*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. ISBN 1-55860-688-2.

[17] R. Nix. Editing by example. In *POPL '84*, 1984.

[18] OpenOffice. Openoffice.org, 2010. URL http://www.openoffice.org/.

[19] SKETCH. Sketch, 2010. URL https://bitbucket.org/gatoatigrado/sketch-frontend/ wiki/Home.

[20] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. In *PLDI '07*, 2007.

[21] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326, 2010.

[22] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. Foster. Path-based inductive synthesis for program inversion. In *PLDI*, 2011.

[23] A. Taly, S. Gulwani, and A. Tiwari. Synthesizing switching logic using constraint solving. In *VMCAI*, pages 305–319, 2009.

[24] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *SIGMOD '09*, 2009.

[25] I. H. Witten and D. Mo. *TELS: learning text editing tasks from examples*, pages 183–203. MIT Press, 1993.